Analysis on Inheritance in Open Source Systems

Edward McNealy

This dissertation was submitted in part fulfilment of requirements for the degree of MSc Advanced Software Engineering

DEPT. OF COMPUTER AND INFORMATION SCIENCES

UNIVERSITY OF STRATHCLYDE

SEPTEMBER 2016

McNealy

DECLARATION

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc Advanced Software Engineering of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself. Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive.

(please tick) Yes [✓] No [ ]

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices) is

I confirm that I wish this to assessed as a Type 1 2 3 4 5 Dissertation (please circle)

Signature:

Date:

August 29, 2016

Abstract

Inheritance is a key component in object-oriented programming, providing many of the benefits that are available through this programming paradigm. These benefits include increased modularity and code reusability, and the freedom to make changes to a single base class and have those changes applied to all who inherit that class. Inheritance also comes with some disadvantages, including a loss of encapsulation when the state of a class is shared among multiple children, and an increase in code complexity. Knowing when to correctly and efficiently use inheritance, and when to consider an alternative, is a valuable skill that any developer should strive towards.

By utilizing established software code metrics as a basis for analysis, this thesis aims to provide detailed observations on the use of inheritance in open source Java applications. These metrics help to reveal information about the depth of inheritance hierarchies, the risks associated with a class having many children and methods, the growth and complexity of complete inheritance hierarchies, and the effects of inheritance on methods implemented in both parent and child classes. This information can be helpful in comparing the ways different open source systems use inheritance.

An analysis tool has been developed for the purpose of investigating inheritance use in open source systems. The metrics mentioned above are used by this tool to analyze different aspects of inheritance and provide data that is easily consumed through charts and graphs. With the help of this tool and the code metrics, examples of both decent and poor design practices regarding inheritance will be revealed.

Acknowledgements

I would like to thank my thesis advisor Dr. Murray Wood of the Department of Computer and Information Sciences at the University of Strathclyde. Dr. Wood was extremely helpful during the process of completing the thesis, contributing many useful resources and helping to push the project in the right direction. Most importantly, he was always available to review my work and provide constructive feedback that kept me thinking on ways to improve.

I would also like to thank my parents, who have always encouraged me to continue learning, and who support me whenever I need help. Finally, I'd like to thank my brothers, for keeping me entertained and distracted (in a good way) during my time in university.

Edward McNealy

Table of Contents

Chapter 1: Introduction to Inheritance 8
1.1 Terminology 10
1.2 Principles of Inheritance 11
1.3 Single Inheritance 14
1.4 Multiple Inheritance 16
1.5 Inheritance in Java 16
1.6 Dangers of Inheritance 22
1.7 Inheritance Summary 24
Chapter 2: Software Code Metrics 26
2.1 Overview on Established Metrics 26
2.2 Depth of Inheritance Tree 27
2.3 Number of Children 30
2.4 Weighted Methods per Class 32
2.5 Code Metrics Summary 33
Chapter 3: Inheritance Inquiry Application 35
3.1 Main Application Software 35
3.2 Eclipse JDT 36
3.2.1 Abstract Syntax Tree 38
3.2.2 Visitors 39
3.3 Implementation Details 43
3.3.1 Elements 43
3.3.2 Builders 45
3.3.3 Tasks 46
3.3.4 Services 47
3.4 Inheritance Inquiry Summary 52
Chapter 4: Analysis 54
4.1 Multi System Analysis 55
4.1.1 DIT Ranges 56
4.1.2 NOC and WMC 60
4.1.3 Inheritance Hierarchy 64
4.1.4 Effects on Methods 66
4.2 Individual System Investigation 71

4.2.1 Ant 71
4.2.2 ANTLR 72
4.2.3 ArgoUML 72
4.2.4 Azureus 72
4.2.5 FreeCol 73
4.2.6 Hibernate 73
4.2.7 JHotDraw 74
4.2.8 JUNG 75
4.2.9 JUnit 75
4.2.10 Lucene 75
4.2.11 Weka 76
Chapter 5: Conclusion and Future Developments 77
5.1 Analysis Results 77
5.2 Future Developments 79
5.3 Closing Thoughts 80
References 81

Figures

Figure 1.1 Figure hierarchy from JHotDraw 15
Figure 1.2 Deadly diamond of death 16
Figure 2.1 TriangleFigure UML diagram 28
Figure 2.2 DIT levels of tools from JHotDraw 28
Figure 3.1 Use of Java Model in Eclipse 37
Figure 3.2 Standard AST workflow process 38
Figure 3.3 Assert.java hierarchy from JUnit 50
Figure 3.4 Taskdef.java from Ant system in Qualitas Corpus 51
Figure 3.5 Inheritance Inquiry build workflow 53
Figure 4.1 DIT Variance in analyzed systems 57
Figure 4.2 DIT ranges for inheritance use 60
Figure 4.3 Inherited Method Risk 62
Figure 4.4 FreeCol.java inheritance hierarchy 64
Figure 4.5 AbstractHandle.java and LocatorHandle.java widths 65
Figure 4.6 Average overridden methods in analyzed systems 71
Figure 4.7 Dialect.java inheritance hierarchy from Hibernate 74

Code Snippets

Code Snippet 1.1 RectangeBasedTool.java example for JHotDraw 13
Code Snippet 1.2 RadiusHandle.java and AbstractHandle.java from JHotDraw 17
Code Snippet 1.3 Method overriding from JHotDraw 19
Code Snippet 1.4 Method extending from JHotDraw 21
Code Snippet 3.1 ASTNode.java from Eclipse JDT 40
Code Snippet 3.2 ASTVisitor.java from Eclipse JDT 41
Code Snippet 3.3 ClassVisitor.java from Inheritance Inquiry 42
Code Snippet 3.4 SystemBuilder.java from Inheritance Inquiry 45
Code Snippet 3.5 IIJob.java and StandardBuildTask.java from Inheritance Inquiry 47
Code Snippet 3.6 Method for building DIT hierarchies 49
Code Snippet 4.1 DecoratorFigure.java in JHotDraw 68
Code Snippet 4.2 Modifier.java in FreeCol 70

Tables

Table 1.1 Versions of analyzed systems 9
Table 3.1 ASTNodes and IBindings used for Inheritance Inquiry 39
Table 3.2 ArgoUML NOC metric from MetricService 51
Table 3.3 DIT levels for Qualitas Corpus systems and JHotDraw 52
Table 4.1 Total classes and percentage of inheritance usage for analyzed systems 55
Table 4.2 DIT Standard for analyzed systems 58
Table 4.3 Acceptable and dangerous DIT levels 59

Chapter 1: Introduction to Inheritance

Inheritance plays a significant role in many aspects of software related technology today, such as artificial intelligence, object-oriented databases, and object-oriented programming (Thirunarayan, 2009). Taivalsaari claims that the ability of objects to inherit from one another in object-oriented programming is considered one of the main features that distinguishes this programming model from others. Many of the benefits achieved through object-oriented programming, such as improved conceptual modeling and reusability, are possible thanks to the concept of inheritance (Taivalsaari, 1996, p. 438).

So what is inheritance? This feature will be explained in full detail shortly, but the basic concept is that inheritance supports the ability of new object definitions to be modeled after ones that have already been defined, and gives these new objects properties of the parent that can be used in the same manner, given additional functionality, or removed altogether (Taivalsaari, 1996, p. 447). This promotes the reuse of code that has already been written, and provides additional benefits, such as the ability to interchangeably use objects that inherit from the same parent.

The use of inheritance can effect objects in many different ways, some beneficial, and some that cause increased complexity and loss of freedom to make compatible changes to existing classes (Snyder, 1986, p. 39). These problems might not always be apparent to a developer, and they might often use inheritance simply because it is available and provides some advantages, while not considering the negative side effects that may follow later in an application's lifetime. In order to better understand how inheritance might affect an application, there are certain code metrics and analysis methods that can be used to provide information on the impact inheritance has on related objects. The major questions to consider when doing this analysis are the following:

- Is there any risk or additional complexity for objects that are deep in an inheritance hierarchy?
- What is the correlation between an object that is inherited many times, and the methods that are implemented in that object?

- How does the hierarchy structure of one object change and grow as more separate objects inherit from the parent?
- What happens to methods that are defined in a parent object, when they are also implemented in child objects?

To help with the process of answering these analysis questions, a tool has been developed using the Java programming language and the **Eclipse JDT**. This application is designed to take Java software systems and parse the code into a tree-like structure that allows for inspecting the sections of code that relate to inheritance use. Then a number of calculations will be made that will provide data that can be analyzed to provide a deeper comprehension on the answer to the proposed questions.

The eleven systems that are to be analyzed are all open source Java projects. All but one of those systems come from the **Qualitas Corpus** (Tempero, 2013), which is a collection of software systems maintained by Ewan Tempero of the University of Auckland, and intended to be used for empirical studies of code artifacts. The final system is **JHotDraw** (Gamma & Eggenschwiler, 2007), which was developed by Erich Gamma and Thomas Eggenschwiler as a way of promoting well-known design patterns, identifying new patterns, and to be an example of a well-designed and flexible framework (Gamma & Eggenschwiler, 2007). The versions of each of these systems can be seen in *Table 1.1*.

System	Version
Ant	1.8.4
ANTLR	4.0
ArgoUML	0.34
Azureus	4.8.1.2
FreeCol	0.9.5
Hibernate	4.2.2
JHotDraw	6.0.1
JUNG	2.0.1
Junit	4.9
Lucene	4.2.1
Weka	3.7.9

Table 1.1 Versions of analyzed systems

The remainder of this introductory chapter will establish the terminology used throughout this thesis and provide details on the history of inheritance in object-oriented programming along with the definition of what inheritance actually means. There will also be information regarding two different types of inheritance available in programming languages today, and explain how inheritance is used in the language chosen for the application supporting the research for this thesis. Finally, there will be a section on some precautions to consider when using inheritance in any application.

1.1 Terminology

Throughout the scope of this paper there are some terms that will be used to refer to certain aspects of programming. When referring to an item that inherits from another, the words *subclass*, *child*, and *descendant* may be used. When referring to an item that is being inherited by another, the words *superclass, parent,* and *ancestor* may be used. The term *type* is used to define what kind of generic object is being used in Java, be it a class or an interface. For example, a *List<String>* is a *List* of objects of the type *String*.

Certain words and terms have been marked in order to recognize their type. When referencing a class or interface, the word is italicized, for example: *java.lang.Object*, or *AbstractFigure*. Method signatures or method names will also be in italics, for instance, *moveBy(int, int)* or *draw*. Reserved keywords in the Java programming language are in bold, such as **extends** or **super**. Applications or systems that were used for the purposes of the analysis are also in bold, like **Hibernate** or **Inheritance Inquiry**.

For examples of an application that uses inheritance, the project **JHotDraw** will be used. **JHotDraw** is an open source project created by Erich Gamma and Thomas Eggenschwiler. This application provides many representations of well-known design patterns that were popularized by Gamma, along with Richard Helm, Ralph Johnson, and John Vlissides. There are also many uses of inheritance throughout the program that might be used to as examples.

There are three design patterns that are mentioned throughout this thesis. These concepts are defined by Gamma et al. in their book *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma, et al., 1995, pp. 139, 175, 325, 331). In order to understand the patterns when they are used, the definitions are provided here.

- **Adapter –** "Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces."
- **Decorator –** "Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."

- **Template –** "Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."
- **Visitor –** "Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."

1.2 Principles of Inheritance

Inheritance is one of the fundamental concepts of object-oriented programming, alongside encapsulation and polymorphism. One of the earliest introductions to inheritance was in the programming language Simula, developed by Ole-Johan Dahl and Kristen Nygaard in the 1960s. This language introduced the concept of "classes" and "objects", and can be seen as the first object-oriented programming language (Dahl, 2004). The technique of inheritance was defined by Dahl and Nygaard as *concatenation*, and could be described as the process of merging attributes from two different components (classes or objects), and the composition of their actions (Dahl, et al., 1972). By allowing new objects to be defined based on existing objects, only those properties that are not already declared in the existing class need to be declared in the new object. The properties of the base object will be available to the new object, and automatically included, without the need for redefinition (Taivalsaari, 1996). Based on this information, inheritance can be defined as the following:

Inheritance is the process through which objects are able to reuse the properties and behavior of already existing objects, thereby limiting the amount of code that needs to be rewritten, and providing a more modular approach to development.

Through the use of inheritance, a class will receive certain methods and properties from another class, allowing the inheriting class to have a similar state and structure as the parent class. Gamma, et al. note that "(w)hen a subclass inherits from a parent class, it includes the definition of all the data and operations that the parent class defines" (Gamma, et al., 1995). Methods and variables that are defined inside a parent class will be available to use in any subclasses, and objects that reference those subclasses can also call methods from the parent.

This is very useful in achieving code reuse throughout an application, as code can be defined in one class, and then simply inherited in others, avoiding multiple definitions of the same code. This type of code reuse is often recognized as “white-box reuse” (Gamma, et al., 1995, p. 31). As noted by Crnogorac et al, the main benefit received through the use of inheritance is the reduced amount of effort needed in implementation, as classes that are already developed and tested can be branched into new classes, and flexibility is obtained since code changes can be limited to only one class in certain circumstances (Crnogorac, et al., 1998, pp. 572-573).

With inheritance now defined, it is important to understand another concept of OO programming: encapsulation. Alan Snyder defines encapsulation as “a technique for minimizing interdependencies among separately-written modules by defining strict external interfaces” (Snyder, 1986, p. 39). Using encapsulation allows a developer to update the implementation details of an object, without having to change the clients that use that object (Micallef, 1988, p. 10). While inheritance can be seen as a white-box, in that the internal details of a class are known to its children, encapsulation can be seen as a black-box. The details of an object’s implementation are unknown to clients, all the client is aware of is the interface that is **public** from that object.

In addition to the reusability benefits of inheritance, another important feature is type substitution. When one class inherits from another, the subclass is allowed to be used in the code in place of the superclass. The benefit afforded through type substitution is that the details of an object’s implementation can be hidden away, and the client that uses that object only needs to be aware that the object is of the desired type. The code also becomes more modular as many different types can be passed as a reference that is asking for one specific type of class. An example of type substitution can be seen in **JHotDraw**. Consider a **JHotDraw** application that requires a tool which can only create *Figures* that are based on a rectangle. The details of how this would work are shown in *Code Snippet 1.1* and explained in the following paragraph.

First, a *RectangleBasedTool* can be created, which **extends** *CreationTool*, and accepts any *object* of the type *RectangleFigure* as a parameter for its constructor. The **super** constructor is called, passing in the *RectangleFigure*. *CreationTool* requires a *Figure* as the parameter, and because

RectangleFigure **implements** *Figure*, this is valid and the class is complete. Now in any **JHotDraw** app, the *RectangleBasedTool* can be used for figure creation. However, unlike a *CreationTool* which can be used with any *Figure*, this new tool will only accept a class that inherits from *RectangleFigure*. Any type of *RectangleFigure* can be substituted as a parameter for this tool, and because the tool is not restricted to one **final** class, it is possible to switch between those types at run-time.

```
public class RectangleBasedTool extends CreationTool {

  // Only accepts classes that inherit from RectangleFigure
  public RectangleBasedTool(DrawingView view, RectangleFigure prototype) {
    super(view, prototype);    // super constructor accepts any Figure
  }
}

public class CreationTool extends AbstractTool {
  private Figure fPrototype;

  public CreationTool(DrawingView view, Figure prototype) {
    super(view);
    fPrototype = prototype;
  }
}

public class RecatangleExampleApp extends DrawingApplication {
  // ....
  @Override
  protected void createTools(JPanel palette) {
    Tool rectTool = new RectangleBasedTool(view(), new RectangleFigure());
    Tool diamondTool = new RectangleBasedTool(view(), new DiamondFigure());
    Tool triangleTool = new RectangleBasedTool(view(), new TriangleFigure());
    // Add tools...
  }
}
```

Code Snippet 1.1 RectangeBasedTool.java example for JHotDraw

In contrast to the benefits of inheritance mentioned above, there are also considerations to make regarding how to safely use inheritance. With inheritance, the implementation details of a subclasses are based on the parent class, forming a contract between the parent and any children. This contract will limit the rate at which a developer can safely make changes to the parent class, as those changes may have unintended side effects on the subclasses (Snyder, 1986, p. 39). Modifications to a parent class will often force the children classes to change as well (Gamma, et al., 1995, p. 31). Because classes using inheritance are defined at compile-time,

another restriction associated with its use is that the implementation details from a parent class are fixed and cannot be changed at run-time (Gamma, et al., 1995, p. 31).

When studying classes in a system, is it important to consider the relationship between those classes. According to Bertrand Meyer, "inheritance represents the *is* relation, also known as *is-a*" (Meyer, 1988, p. 498). Using **JHotDraw** as an example, there is a *ConnectedTextTool* class which **extends** *TextTool*, so it can be said that "every *ConnectedTextTool* is a *TextTool*". This relationship is a contrast against the one provided by composition. Object composition is an alternative to inheritance, in which a class contains a reference to another object (preferably an interface), and any actions required by the containing class are delegated to the composition object. The relationship with composition is a *has-a* relation. Using this alternative provides a "more workable and flexible extension mechanism" where objects can be easily added or removed at run-time of an application (Gamma, et al., 1995, pp. 43-44).

There are generally two types of inheritance: *single inheritance* and *multiple inheritance*.

1.3 Single Inheritance

With single inheritance, a class is only able to inherit information from one parent. That parent can then also inherit from another, and so on for each subsequent parent. In Java, this inheritance path can theoretically continue on an infinite amount of times, although in a study on the maximum depth of inheritance, Dr. Heinz Kabutz found that a *StackOverflow* error was thrown after compiling and running a class with 61 parents, and that a class with 1001 parents would not compile (Kabutz, 2006).

In examination of JHotDraw, the figure classes can be used as an example on the use of inheritance in a Java application. Most of the visual components of a JHotDraw application will be based around some type of *Figure*. All of these classes will implement the *Figure* interface, which defines the contract for the basic methods needed to display, move, connect to, and modify the component. There is an abstract base class for figures available for use through inheritance, called *AbstractFigure*. This contains default implementations of many of the methods defined by the *Figure* interface, as well as some abstract methods that child classes will be expected to implement. Because *AbstractFigure* already **implements** the *Figure*

interface, any subclasses of *AbstractFigure* will naturally implement *Figure* as well, through inheritance. This causes any classes using this inheritance to also be of the *Figure* type, creating the possibility for type substitution of those classes wherever a *Figure* interface is required.

There are additional abstract figures that extend the base *AbstractFigure* class, such as *CompositeFigure* and *AttributeFigure*, adding additional functionality while still being flexible for changes. Whenever a figure class needs to be developed, it should inherit from one of these classes. An additional interface for joining figures, *ConnectionFigure*, is also inheriting from the *Figure* interface. The hierarchy structure for *Figure* can be seen in *Figure 1.1*, with the *Figure* interface being the root node of the tree. In this diagram, interfaces are gray, and classes are blue. This hierarchy was generated through the use of services available with the tool developed for the purpose of this research, which will be discussed in further detail in Chapter 3. Note that the labels for the selected 5 nodes were added after the graph's creation, and only provided for the 5 objects mentioned here, ignoring the remaining *Figure* classes.

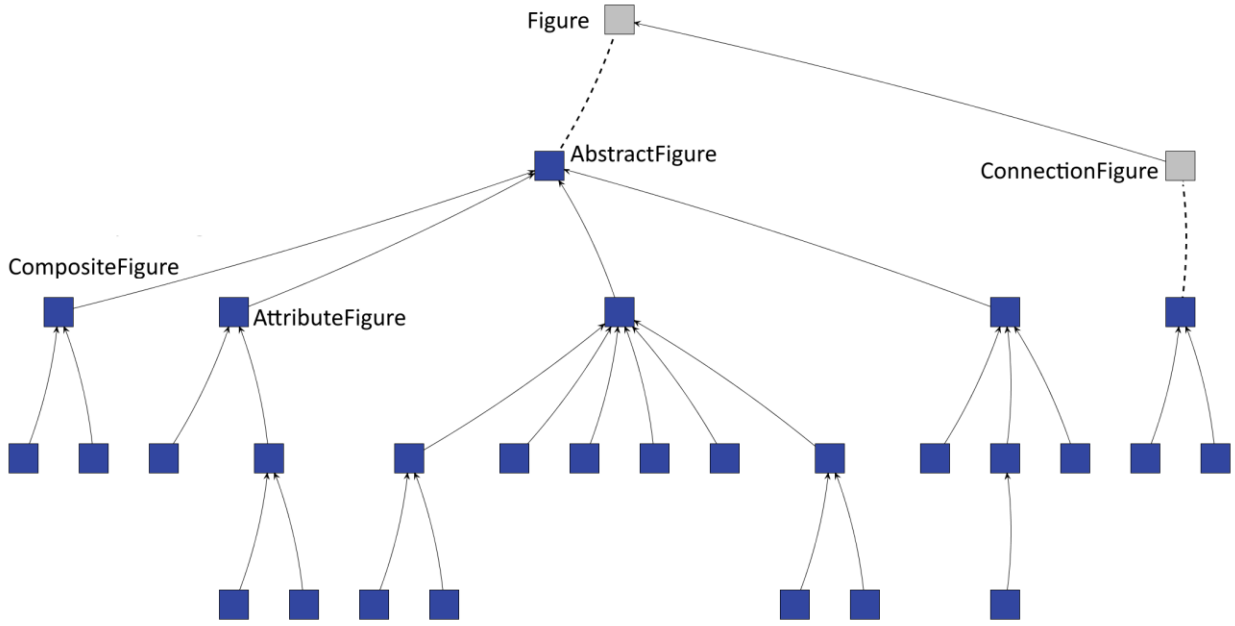

Figure 1.1 Figure hierarchy from JHotDraw

1.4 Multiple Inheritance

The definitive difference between single and multiple inheritance is simple: multiple inheritance allows a class to inherit from an arbitrary number of parents. In actual programming practice, the difference can be much more complicated, as are some special considerations that must be taken into account. One of the main problems that arises from allowing multiple inheritance is termed the "*deadly diamond of death*" by Robert Martin. This results from a situation in which a class *A* is the top-level of a hierarchy, with two classes *B* and *C* inheriting from *A*, and class *D* inheriting from *B* and *C*. When *B* and *C* override a method *foo* defined in *A*, and *D* does not provide an overridden implementation, then the version of *foo* that will be used is unclear (Martin, 1997). There are ways to resolve this issue, such as feature renaming (Meyer, 1988, pp. 535-540), as well as individual solutions provided for those languages that do support multiple inheritance.

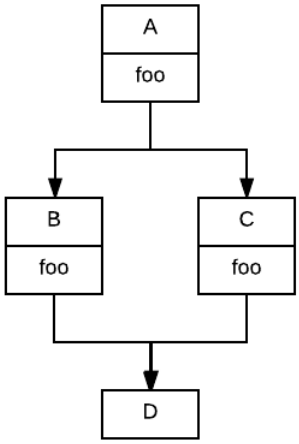

Figure 1.2 Deadly diamond of death

1.5 Inheritance in Java

Before inheritance in the Java programming language can be explained, access modifiers must also first be understood. An access modifier is the rule used by Java to declare how methods and variables can be given or denied access to other classes. When decorated with the **private** modifier, an item will be only available to the class that declares it. The **public** modifier is the opposite, as **public** items are available at a global scope. The **protected** modifier gives access of an item to a subclass, or to another class in the same package. When no modifier is given, a default known as package-private is used, which gives access of an item to other classes in the same package. Inheritance in Java relies heavily on the **protected** access modifier in order to allow subclasses to use methods and variables that are declared in the parent.

In the Java programming language, inheritance of a class is declared using the **extends** keyword. By saying that class *A* **extends** class *B*, it can be inferred that class *B* is the parent class, and class *A* is the child class. *A* will now be able to make use of any properties from *B*,

provided they are decorated with the **protected** or **public** keyword. Properties in the child class can override ones that are declared in the parent. This is achieved by using the same name or method signature as a variable or method in the parent. The *@Override* annotation can be used to ensure at compile-time that a method is in fact overriding another. Inheritance is the way that parents can share their state with any of their children. That state can be **protected**, and only available between the parent and its children, or **public**, and available to any client that uses either the parent or child.

To access a property that is declared in a parent class, the child class can make use of the **super** keyword. When creating a constructor for a subclass, the **super** keyword can be used to first call the parent class constructor. This is required when a parent has a constructor with parameters and does not provide a no-argument constructor, or the no-argument constructor is **private** to the parent. A JHotDraw example of using **super** in a constructor can be seen in *Code Snippet 1.2*. When the *RadiusHandle* calls *super(owner)*, the constructor in *AbstractHandle* is invoked.

```
public abstract class AbstractHandle implements Handle {

  private Figure fOwner;

   * Initializes the owner of the figure.
  public AbstractHandle(Figure owner) {
    fOwner = owner;
  }
  //...
}

public class RadiusHandle extends AbstractHandle {

  private RoundRectangleFigure fOwner;

  public RadiusHandle(RoundRectangeFigure owner) {
    super(owner);
    fOwner = owner;
  }
  //...
}
```

Code Snippet 1.2 RadiusHandle.java and AbstractHandle.java from JHotDraw

Methods that are overridden by a subclass are the ones that will be accessed when that subclass is used as a reference in another class. To explain this further, the *RadiusHandle* class

from JHotDraw can again be examined. This class inherits from *AbstractHandle*, which provides a default implementation for the *draw()* method, used to display the handle on the GUI. The *AbstractHandle* class draws handles with a rectangular shape. The *RadiusHandle* is used to create handles who have a circular shape. Because of this, the default implementation of *draw()* cannot be used in *RadiusHandle*. Instead, this class will override the *draw()* method. Wherever a reference to a *RadiusHandle* is used, this overridden method will be used in place of the *AbstractHandle draw()* method. In JHotDraw, Handles are created within the drawing view by looping through each instance of Handle and calling the *draw()* method. The code in *Code Snippet 1.3* shows the details of how this works.

```
public abstract class AbstractHandle implements Handle {

  public AbstractHandle(Figure owner) {
    // ...
  }

  //Default draw method Template method
  @Override
  public void draw(Graphics g) {
    Rectangle r = displayBox();     // Gets the display bounds for the handle

    g.setColor(Color.white);
    g.fillRect(r.x, r.y, r.width, r.height);

    g.setColor(Color.black);
    g.drawRect(r.x, r.y, r.width, r.height);
  }
}

public class RadiusHandle extends AbstractHandle {

  public RadiusHandle(RoundRectangleFigure owner) {
    super(owner);
    // ...
  }

  @Override
  public void draw(Graphics g) {
    Rectangle r = displayBox();     // Gets the display bounds for the handle

    g.setColor(Color.yellow);
    g.fillOval(r.x, r.y, r.width, r.height);

    g.setColor(Color.black);
    g.drawOval(r.x, r.y, r.width, r.height);
  }
}

public class StandardDrawingView extends // ...
                                                                    {
  /**
   * Draws the currently active handles.
   */
  @Override
  public void drawHandles(Graphics g) {
    // All the handles
    // In this case, the handles we are using are RadiusHandles
    Enumeration<Handle> k = selectionHandles();
    while (k.hasMoreElements()){
      // Calling the draw() method in RadiusHandle
      (k.nextElement()).draw(g);
    }
}
```

Code Snippet 1.3 Method overriding from JHotDraw

In addition to simply overriding a method, a subclass is able to extend a method from its superclass. For cases when a subclass wants to continue using a superclass's method, but also needs to provide additional functionality, the subclass might extend the method functionality. The method will be overridden in the subclass, but still call the **super** implementation (Lorenz & Kidd, 1994, p. 69). This is similar to the way a subclass might include a call to the parent's constructor when implementing its own constructor. However, the **super** call is not restricted to being the first statement in the extended method, and can be called at any time inside the method body.

Code Snippet 1.4 shows an example of how method extending is accomplished. A *BorderDecorator* is a type of *Figure* that uses the **Decorator** design pattern to add a border around any other *Figure*. This is done by first rendering the actual *Figure* using the call to *super.draw(g)*, which calls the *draw()* method in *DecoratorFigure*. After this, *BorderDecorator* continues with the extended functionality of rendering a border around the boundaries of the *Figure* to be decorated.

```
public class BorderDecorator extends DecoratorFigure {

  public BorderDecorator(Figure figure) {
    super(figure);
  }

   * Draws the figure and decorates it with a border.
  @Override
  public void draw(Graphics g) {
    Rectangle r = displayBox();
    super.draw(g);
    g.setColor(Color.white);
    g.drawLine(r.x, r.y, r.x, r.y + r.height);
    g.drawLine(r.x, r.y, r.x + r.width, r.y);
    g.setColor(Color.gray);
    g.drawLine(r.x + r.width, r.y, r.x + r.width, r.y + r.height);
    g.drawLine(r.x , r.y + r.height, r.x + r.width, r.y + r.height);
  }
}

public abstract class DecoratorFigure
                extends AbstractFigure
                implements FigureChangeListener {

   * The decorated figure.
  protected Figure fComponent;

   * Constructs a DecoratorFigure and decorates the passed in figure.
  public DecoratorFigure(Figure figure) {
    // ...Sets figure = fComponent and sets up listeners
  }

   * Forwards draw to its contained figure.
  @Override
  public void draw(Graphics g) {
    fComponent.draw(g);
  }
}
```

Code Snippet 1.4 Method extending from JHotDraw

All classes in Java have as their top level ancestor the *java.lang.Object* class. This class does not need to be explicitly declared; instead it is automatically assigned by Java. *Object* provides all classes in Java with some standard methods, such as *equals()* and *toString()*. While these methods have default functionality, it is usually helpful to override these methods and provide a custom implementation, with *equals()* determining if an object's variables are the same as another, and *toString()* providing a more informative description of a class. The inheritance structure that is provided by *Object* to all classes in Java ensures that every class will have *Object* as the element at the very top level of their hierarchy. Because of this, the hierarchy

level for *Object* can be noted as 0, with all other classes starting with the hierarchy depth being at 1, unless they inherit from another class.

Java does not allow for a class to use multiple inheritance, so this will not be a concern for the scope of this research. The closest that Java comes to multiple inheritance is through the design of interfaces. A class is allowed to implement more than one interface, which will create a contract on that class that expects the class to contain certain method signatures. Since interfaces only contain abstract contractual method signatures and cannot be initialized as an object, they do not typically contain the dangers associated with multiple inheritance. In Java 8, interfaces are allowed to contain **default** methods, which do provide an implementation in the interface. This can introduce a diamond problem, however classes with this problem will cause compiler errors in order to prevent this.

1.6 Dangers of Inheritance

While inheritance is beneficial to the structure of an OO system, there are also drawbacks that are included, and considerations that must be made towards the effects against other components of the system. Snyder explains that "(p)ermitting direct access to inherited instance variables weakens one of the major benefits of object-oriented programming, the freedom of the designer to change the representation of a class without impacting its clients" (Snyder, 1986). This means that because classes that inherit from another have access to properties and methods from their parent, and use them among their own local properties and methods, changes to the parent can result in unexpected changes to the children. For an example of this, consider an abstract class that is implementing a **Template** method. A **Template** method delegates part of an algorithm to subclasses. The work done in the delegated steps usually depends on the operations being performed in the skeleton, which is defined in the parent. If those parent operations are altered, the actions done by the children may have unintended side-effects. Developers need to consider this before making changes to the parent, and include tests that ensure the behavior is unchanged after the **Template** is altered.

A problem with inheritance identified by Snyder is that "the instance variable operations defined on a class for the benefit of its descendants may not necessarily be appropriate for

users of instances of the class" (Snyder, 1986). In order to correct this problem, Java provides access modifiers that should be used in the appropriate situations. The use of "getters" and "setters" as operations to access instance variables from a class and its parent should be avoided when possible. Allen Holub explains that while the use of these operations is somewhat standard in many Java applications, it is in fact not quite a good object-oriented practice (Holub, 2003a). Inheritance must work alongside encapsulation in order for a system to follow OO guidelines. As the primary goal of encapsulation is to contain the scope of an object to solely that object, and prevent other objects from being aware of any details of the variables in a class, getters and setters should be used sparingly. According to Holub, "you should avoid getter and setter functions since they mostly provide access to implementation details" (Holub, 2003a). In an article discussing the issues of getters and setters, Greg Jorgensen states that "getters and setters should be avoided because they break the encapsulation OOP offers" (Jorgensen, 2008).

Snyder notes that, when using inheritance and a local instance variable and a variable inside a parent have the same name, then changing the name of the parent variable will potentially affect the behavior of the child. He also notes that raising an error at compile-time for instance variables in a child and parent with the same name would not be appropriate, as these variables should be separate instances (Snyder, 1986). In Java, this is avoided because access to variables in a parent class are only available when that variable has the **protected** or **public** access modifier, and unless the variable is accessed through the use of the **super** keyword, it is assumed that the variable is the local instance.

Inheritance is often used to define a "base" class that contains functionality that is common among many classes. Those classes can then extend the base class to gain that functionality, avoiding the problem of rewriting code in multiple locations. However, this results in what is recognized as the "fragile base-class problem", which means that modifications to a parent may seem to have no adverse effect on their children, but can cause problems in those classes. In the article "Extends is Evil" by Holub, he states that "(y)ou can't tell whether a base-class change is safe simply by examining the base class's methods in isolation; you must look at (and test) all derived classes as well" (Holub, 2003b). He also notes that you must examine other

classes that contain references to both the parent and the child classes that are being modified, since this coupling between classes can result in unintended side-effects in these classes as well. The solution to his proposal that "extends is evil" is to avoid inheritance through superclasses with the **extends** keyword, and instead prefer to use inheritance through interfaces, with the **implements** keyword. This is directly related to OO design principle of "*program to an interface, not an implementation"* (Gamma, et al., 1995). The advantages gained through the use of interface inheritance are the following, again provided by Gamma, et al.:

- "Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect."
- Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface."

One way to achieve this interface based inheritance is by using the **Adapter** design pattern. The **Adapter** will "convert the interface of a class into another interface clients expect" (Gamma, et al., 1995). Object composition can be used to delegate the actions of the **Adapter** to another class, circumventing the need for the **Adapter** to inherit from another class. In this way, the **Adapter** can still adhere to the interface that the client expects, without inheriting any extra properties through a parent class.

1.7 Inheritance Summary

Inheritance can be defined as *a process through which objects are able to reuse the properties and behavior of already existing objects*, which is also known as white-box reuse. Some of the features available through the use of single inheritance is the sharing of properties and behavior from one class with many children, and the ability to substitute one type of subclass with another, as long as they have the same parent. Inheritance use can also include inconvenient side effects. This includes restrictions on the changes that can be made to a parent class, as those changes can alter the behavior of any subclasses, and may require those subclasses to also be modified. There might also be conflicts with names of properties and methods in a child class that match ones defined in the parent. However, this can be handled

by the programming language used, such as Java's use of the **super** keyword to distinguish between child and parent properties and methods.

Chapter 2: Software Code Metrics

The software quality-measurement values that are used in this research will be described in this chapter. This will include the definition for each code metric, viewpoints that should be considered before making use of a metric, and information on how to interpret each metric. While interfaces in Java are able to use inheritance, and are even able to inherit from multiple parents, they are unable to have concrete definitions of methods, and cannot be initialized as an *Object*. Because they have no effect on the inheritance hierarchy of *Objects*, **interfaces will not be considered** in any of the code metrics used throughout the scope of this project.

2.1 Overview on Established Metrics

In order to measure the degrees to which a software system meets certain qualifications, code metrics can be used to extract information about the system. When inspecting an element in an application to determine the quality of design choices regarding the use of that element throughout the scope of the application, there are two important metrics that may be considered. These metrics are the *depth of inheritance* and the *number of children*. These metrics were conceptualized by Shyam Chidamber and Chris Kemerer, and are part of a software metrics suite consisting of four other metrics, for a total of six (Chidamber & Kemerer, 1994). Another useful metric is the *weighted methods per class*, which will be helpful in this thesis by revealing information about how class inheritance impacts the amount of methods cascading down the inheritance tree.

The *Figure* classes used in **JHotDraw** are good examples for showing the use of inheritance in an application. Since **JHotDraw** is a graphical-based application used for drawing diagrams and shapes, there are many instances of *Figures* that can be used. All *Figures* in **JHotDraw** extend the *AbstractFigure* class, making this class the top-level of their inheritance hierarchy. The *AbstractFigure* implements the *Figure* interface, which forms a contract that declares the methods that all *Figure* classes will be expected to implement. *AbstractFigure* will be useful in explaining the details on *number of children*.

In the same manner that Chidamber and Kemerer make assumptions about the distribution of properties for their metrics, there is a fact that is apparent in this thesis (Chidamber & Kemerer, 1994). For each of the analyzed systems that are in use for the purposes of this research, we can be certain about the fact that the system *does* make some use of inheritance, and therefore is composed of both classes who have no parents (by making no use of inheritance at all, or by being at the root of the inheritance hierarchy tree) and classes that extend another class (using inheritance).

2.2 Depth of Inheritance Tree

A basic definition for the *depth of inheritance tree* (DIT) for an application that only allows for single inheritance would be the amount of elements that a single element is inheriting from, all the way to the root of the inheritance hierarchy tree. According to Chidamber and Kemerer, "(t)he depth of a node of a tree refers to the length of the maximal path from the node to the root of the tree" (Chidamber & Kemerer, 1994, p. 480). The DIT is a useful tool for measuring the complexity of a class and the application that class is a component of by providing a strong overview of the various levels of inheritance hierarchies in an application.

When the depth of a class's hierarchy tree is realized, a developer can determine how complex that class might be. With a deep hierarchy, there may be many classes that have to be navigated through before a property or method can be reached. Taking the *TriangleFigure* from **JHotDraw** as an example, we can inspect the amount of work necessary to trace a function for this class. A developer might be interested in seeing how this *Figure* is moved around the screen. Searching through *TriangleFigure*, there is no method having to do with movement, so the superclass *RectangleFigure* must then be inspected. This class contains a *basicMoveBy(int,*

int) method, but the name implies that there is more to this function. However, in the next superclass, *AttributeFigure*, there is again no definition for a method with behavior that would move a *Figure*. So one more superclass, *AbstractFigure*, must be examined. Here we can find the abstract definition for *basicMoveBy()*, as well as a method *moveBy(int, int)*. In this method statement, the *basicMoveBy()* method is called. Taking one final step into the *Figure* interface will show that the *moveBy(int, int)* method is defined here, so the developer knows that the *AbstractFigure* implementation is where the method originates from. This process can be viewed in the UML diagram in *Figure 2.1*. Tracing a method through an inheritance hierarchy can be simplified in some ways, such as using breakpoints to step through code execution while debugging, or through the use of an IDE such as Eclipse, which allows for quickly jumping to an implementation.

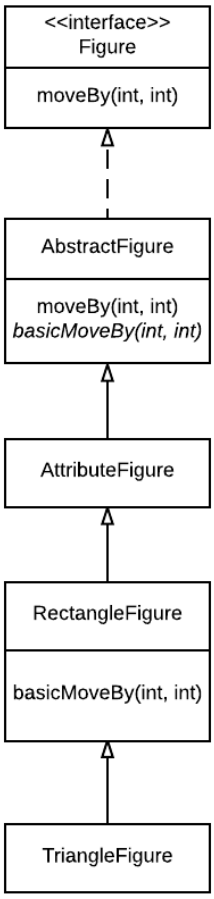

Figure 2.1 TriangleFigure UML diagram

As the root of all classes, *java.lang.Object* has a depth of inheritance of 0. So in the case of a class that does not extend any other classes, the depth of inheritance would be 1. A subclass of that class would have a depth of inheritance of 2, and the number continues to increase by one for each subsequent child. Since we know that each system that is being analyzed for this project is using inheritance in some way, it can be recognized that there will be classes whose DIT is greater than 1. Since the inheritance trees will consist of branching nodes, it can also be realized that there will be at least two classes who share the same level of depth of inheritance. For example, in **JHotDraw** the *AbstractTool* is a base class that does not inherit from any other class, so its DIT is 1. Both *CreationTool* and *SelectionTool* inherit from *AbstractTool*, so they have a DIT of 2.

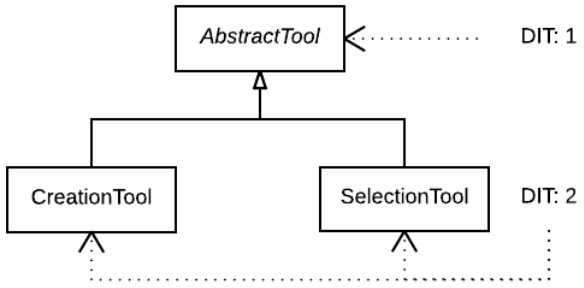

Figure 2.2 DIT levels of tools from JHotDraw

There are three main viewpoints to consider in regards to depth of inheritance. According to Chidamber and Kemerer (Chidamber & Kemerer, 1994, p. 483) those viewpoints are:

1) "The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior."
2) "Deeper trees constitute greater design complexity, since more methods and classes are involved."
3) "The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods."

Methods and variables declared in a base class will be available to any classes that extend the base, provided those methods or variables are **protected**, **public**, or package-private (and the inheriting class is in the same package). As a class becomes deeper in an inheritance hierarchy, it becomes more complex because of the additional methods and properties already defined in each of its parents, all the way to root class. This increased complexity is the trade-off that results from the white-box reuse that inheritance provides.

In the analysis of the depth of inheritance levels in a project, the DIT number provides information about how well the project is making use of the OO fundamental concept of inheritance. A lower number implies that a class is less complex, but possibly not taking advantage of minimal code reuse through inheritance. A higher number implies that the project is exploiting the benefits of inheritance code reuse, but that there will most likely be an increase in complexity and possible code errors (Naboulsi, 2011). While there is no concrete ideal level for depth of inheritance, many suggestions are that the limit on the DIT should be no greater than 5. (Kabutz, 2006), (Tandon, 2010) A pilot study conducted by Harrison et al with the assistance of final year BSc. Computer Science students in the University of Southampton reached a similar conclusion. Their study on the effect of code changes to classes using inheritance found that between 3 and 5 levels of inheritance is ideal (Harrison, et al., 2000, p. 6).

2.3 Number of Children

The *number of children* (NOC) can be defined as the amount of elements that directly inherit from another single element, or the "(n)umber of immediate descendants of the class" (Chidamber & Kemerer, 1994). According to Radu Marinescu, the metric for number of children "represents the number of immediate subclasses subordinated to a class in the class hierarchy" (Marinescu, 1998, p. 3). Ramanath Subramanyam and M.S. Krishnan described the number of children as "a count of the number of immediate child classes that have inherited from a given class" (Subramanyam & Krishnan, 2003, p. 298). Every time that a class inherits from another class, the number of children for the superclass increases. In contrast to the depth of inheritance, which describes how deep a hierarchy tree travels from the root node (or the height of the hierarchy), the number of children measures the width of a hierarchy, or how far the branches of the tree will spread.

Chidamber and Kemerer again give three viewpoints on the NOC metric (Chidamber & Kemerer, 1994, p. 485):

1) "Greater the number of children, greater the reuse, since inheritance is a form of reuse."
2) "Greater the number of children, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of subclassing."
3) "The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class."

By allowing many classes to inherit from one parent, the amount of code needed can be reduced, as methods that are common between similar classes is contained in the parent. When the number of children for a class reaches a high amount, this can signify that more abstraction may be needed in the children classes, and perhaps more abstract classes are needed to bridge between the parent and the many children. Arti Chhikara and R.S Chhillar state that "if there are a large number of children of a class, then the abstraction level of that

parent class is reduced." (Chhikara & Chhillar, 2012, p. 365) They recognize this as an indication of the misuse of inheritance. The high number of children will also have an impact on the effectiveness of testing against that class, as the scope that the methods reach is wider than the scope of those classes who have fewer descendants.

The metric for the number of children only includes the immediate classes that inherit from a parent. This means that the graphs of the hierarchy trees for NOC will only extend two node levels deep. The width of these trees, or the number of branches extending from the root node, can be any number greater than 0 (meaning there will be at least 1 subclass).

Through the analysis of the number of children for each class in a project, it can be determined if that project is making good use of the concept of inheritance. If there are many classes who have children, then it is obvious that there are many classes who are inheriting from another class. Marinescu notes that this metric is helpful in determining the misuses of subclassing, and that this may call for a restructuring of a class's hierarchy. He determined that there are two situations in which a high NOC value may prompt a redesign (Marinescu, 1998, p. 3):

- *Insufficient exploitation of common characteristics* – When there are multiple classes who share similar functionality, such as graphical-based classes, or repository classes that access a database, there is a high potential for reuse. If there is a pattern of classes with these characteristics who have a high number of children, then the system may need to be refactored to have some of the common functionality abstracted to another class to be placed higher in the hierarchy.
- *Root of the class hierarchy* – When the root class in a hierarchy has a very large amount of children compared to other classes in the system, refactoring should be considered for that class. The hierarchy may currently be represented as a *tree*, where there is one class at the root, with many children branching off from that root. Instead, it may be wise to consider splitting this large tree into a *forest*, where there will be a number of trees with a smaller amount of branches. This could result in less complexity, as the trees are smaller and easier to navigate and test.

The studies done by Chidamber and Kemerer suggested that classes usually have only a small number of children, with a small deviation of classes who are inherited by many children. Two explanations given for this are that inheritance is not being considered during the planning and designing of new classes, and that there is a lack of communication between programmers of different classes, resulting in missed opportunities for code reuse (Chidamber & Kemerer, 1994, p. 486).

2.4 Weighted Methods per Class

The metric of *weighted methods per class* (WMC) for an object provides the number of methods defined in a class. Chidamber and Kemerer provide the following formula for determining the WMC:

- "Consider a Class C_1 with methods $M_1,\ldots,M_n$ that are defined in the class. Let $C_1,\ldots,C_n$ be the complexity of methods. Then: $WMC = \sum_{i=1}^{n} c_i$ "

In addition to this formula, Chidamber and Kemerer provide 2 key viewpoints on the use of the WMC metric:

- "The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class".
- "The larger the number of methods in a class the greater the potential impact on children, since children will inherit all methods defined in the class".
    - Note: In Java, and for the purposes of this research, this is not always the case, as methods can be declared as **private**, restricting access to any other class.
- "Classes with large number of methods are likely to be more application specific, limiting the possibility of reuse".

Classes with a high method count become much more complicated, and require more time to work through. A developer who is unfamiliar with a class will need to spend time analyzing the methods in that class in order to understand how they relate to the application. As the method count grows, it is likely that those methods will only be useful within the scope of the

application they are defined, unless they are specifically designed to be a part of an external library or application programming interface (API). In a study on detecting design flaws detection in large scale systems, Marinescu notes that "classes with very high *WMC* values are *critical in respect of the maintenance effort* and they might be therefore redesigned in order to reduce the overall complexity level of the class" (Marinescu, 1998, p. 2). His study suggests that in order to reduce WMC, a class can be split into multiple classes.

Most importantly for this research, methods in a class can have a direct influence on the complexity of any subclasses. When one class uses methods that are defined in another class, those classes form a relationship, or become coupled (Chidamber & Kemerer, 1994, p. 479). As more classes have access to an increasing number of methods, there will be more coupling between objects. This will reduce the modularity of an application and increase its complexity, as it may become difficult to trace the origin of method definitions. This might also discourage encapsulation, as objects are sharing their state and allowing access of methods to more objects.

When calculating the WMC of a class, the value for the metric will increase by one for each method defined within that class. This includes any getter or setter methods that are used for accessing class variables. In addition to any method implementations specific to a class, the WMC also includes any overridden methods defined in a parent class. These methods count as ones unique to the implementing class. However, any methods that are defined in a superclass will *not* be included in this metric. The WMC is limited to only those methods in the class which is being analyzed. When creating hierarchies for each of the classes in a project, the method count will then be used to provide details on how those inherited methods affect subclasses. These guidelines for calculating the weighted method count are based on those used by the Project Analyzer developed by Aviosto (Aviosto, 1997).

2.5 Code Metrics Summary

Chidamber and Kemerer have defined a suite of metrics that can be helpful in providing information on the quality of software. The metrics that are important to the purpose of this

thesis are the depth of inheritance tree, the number of children, and the weighted methods per class.

As the DIT of a class increases, the behavior of that class can become more difficult to control, and indicates a higher number of methods and superclasses are involved. The potential reuse of the inherited methods increases as well, which can mean less code being written, as well as more coupling between classes.

A high number of children indicates a large amount of reuse, but can also be an indication that classes are not being designed properly, as common characteristics of the children could possibly be refactored to another class higher in the hierarchy. In addition, classes with many children might require more testing of the methods defined in those classes.

The WMC helps to keep track of the amount of methods that each class has defined, which becomes useful in determining how the method count will be affected by classes using inheritance. Coupling of methods between classes can have an impact on the complexity of those classes.